MCTS Playouts Parallelization with a MPPA Architecture

Aline Hufschmitt, Jean Méhat, Jean-Noël Vittaut

MCTS Playouts Parallelization with a MPPA Architecture

Aline Hufschmitt and **Jean Méhat** and **Jean-Noël Vittaut**
LIASD - University of Paris 8, France
{alinehuf,jm,jnv}@ai.univ-paris8.fr

Abstract

We present a study of the use of a *Multi-Purpose Processor Array* (MPPA) architecture for the parallelization of MCTS algorithms applied to the field of *General Game Playing*. We evaluate the constraints imposed by this architecture and show that the only feasible MCTS parallelization on MPPA is a *leaf parallelization*. We show that the MPPA provides good scalability when increasing the size of the communications which is useful when using synchronous communications to send large sets of game initial positions to be processed. We consider two approaches for the calculation of playouts: the distributed computing of a playout on each cluster and the calculation of several playouts per cluster; the second approach gives better results. Finally, we describe experiments concerning the thread management and present a surprising result: it is more efficient to create new threads than to synchronize permanent threads.

1 Introduction

General Game Playing (GGP) is a branch of Artificial Intelligence with the aim of achieving versatile programs capable of playing any game without human intervention. These programs must be able to analyze the rules of an unknown game, to understand the goals, to discover the sequences of moves leading to victory and to play with expertise.

To find out what moves to play, different techniques of tree search have been developed to explore positions of the game and different branches of possibilities that depend on the selected moves. The root of the tree represents the initial state of the game. The goal is to reach one of the leaves, corresponding to the end of the game, the score of which is the highest possible for the player.

Currently, the best GGP players use variants of *Monte Carlo Tree Search* (MCTS) combining the construction of a game tree with game simulations (playouts) [Browne *et al.*, 2012]. UCT, the most used MCTS algorithm consists in four phases: selection of a path in the game tree according to some policy; expansion of the tree with the creation of a new node; game simulation playing random moves until a terminal position is reached; back-propagation of the playout results to update node's evaluation. The number of playouts done by MCTS is a key factor for the quality of the evaluation of a game tree node [Kocsis and Szepesvári, 2006].

Former GGP players have used Prolog unification to interpret the language describing the rules. These calculations were slow [Schiffel and Björnsson, 2014]. Recently, the use of *Propositional Networks* (propnets) has allowed an important improvement in playout computation speed. A Propositional Network [Schkufza *et al.*, 2008] is a graph representing game rules as a logic circuit with AND, OR and NOT gates and transitions that represent the passage from one game state to another. At each game step, the current position is set at the input of the circuit. The signal is propagated and, at the output, a flag indicates whether this position is terminal. If it is, the player's scores are accessed through flags; otherwise, the list of legal moves is available at the circuit output; the moves chosen by the players are set at the input of the circuit and the signal is propagated to get the next state. A transition consist of setting the next state as the current position.

To increase the number of playouts made in a given time, various approaches of parallelization of MCTS have been implemented on machines with multi-core and/or multi-threads CPU used alone or networked to provide more computing power. Some of these techniques are applied to *General Game Playing* [Méhat and Cazenave, 2011a; Finnsson, 2012] but none, to our knowledge, realizes this parallelism on a player using a *propnet*. The important acceleration of playout computation provided by the use of propnets brings new conditions for parallelization of UCT because communication and synchronization times become significant.

In this article, we study the parallelization of MCTS using *propnets* to interpret game rules on a *Multi-Purpose Processor Array* (MPPA), an architecture dedicated to many-core processing, marketed since 2013 by the Kalray Company (Essonne, France). This recent architecture has barely been tested on practical applications, thus we explore its possibilities and evaluate its limitations.

Section 2 describes the different parallelization techniques proposed in the literature. Section 3 presents the MPPA architecture. Section 4 presents the feasible parallelization on MPPA. Section 5 evaluates the scalability using communications of different size. Section 6 establishes the best approach to compute the playouts. Section 7 compares different ways of synchronizing threads inside clusters.

2 Parallelization approaches of MCTS

In this section, we present the different approaches for parallelizing the MCTS.

Tree parallelization [Gelly *et al.*, 2006] involves several processes in the construction of a single game tree in a shared memory. Threads perform the selection, expansion, simulation and back-propagation phases independently. A global mutex used to protect the tree from concurrent updates creates a bottleneck, so improvements consist in using local mutexes or a lock free algorithm [Enzenberger and Müller, 2010]. *Virtual losses* avoid the selection of the same node by all the threads [Chaslot *et al.*, 2008].

On a distributed memory system, *root parallelization* [Cazenave and Jouandeau, 2007] develops different trees from the same game position, usually on different machines. The different evaluations are collected and combined to choose the best move [Méhat and Cazenave, 2011b; Soejima *et al.*, 2010]. An improvement consists in synchronizing the evaluations of the top nodes at regular intervals to direct the search using the shared information [Gelly *et al.*, 2008]. A drawback is that the same nodes have to be generated on different machines.

With *leaf parallelization* [Cazenave and Jouandeau, 2007], a single tree is constructed by a master process. It executes the selection and the expansion phases and sends the positions to be explored to slave processes calculating the playouts. The back-propagation phase is performed by the master process using the score returned by each slave. Asynchronous communications provide good scalability [Cazenave and Jouandeau, 2008]. This technique is less effective than *root or tree parallelizations* because it suffers from limitations caused by the master process, which is alone to perform the selection phase, and communications needed for each playout [Soejima *et al.*, 2010]. To minimize communication costs, [Finnsson, 2012] has proposed to realize multiple playouts per position but some unworthy positions are then over-exploited. [Chaslot *et al.*, 2008] have proposed to stop a group of playouts that does not look promising to search again from another position but this approach does not provide good scalability.

In the *UCT-Treesplit* algorithm [Graf *et al.*, 2011; Schaefers *et al.*, 2011] the nodes of a single tree are spread over distributed memories. The machine on which a node is stored is selected using a hash key of this node. During the selection phase, the traversal of edges between nodes stored into different machines is implemented by a request. If a machine receives a request for an existing node, it takes over the selection; if not, it creates this new node and launches a playout. After the playout, an update message has to be sent to all the machines storing parent nodes for the back-propagation. A drawback is the importance of communications during selection and back-propagation.

3 MPPA architecture

The MPPA-256 chip is a multi-core processor composed of 256 processing cores (PC) organized in a grid of 16 clusters connected through a high-speed *Network-on-Chip* (NoC). It is the first member of the MPPA MANYCORE family, the others reaching up to 1024 processors in a single silicon chip.

Each cluster contains 2MB of memory shared by the 16 PCs and a system core. The system core, running NodeOS, supervises the scheduling and execution of tasks on PCs and data transfers while the 16 PCs are dedicated to application code. Each PC can execute its own code, stored in the shared memory.

The limitation of a cluster memory to 2MB is a major constraint in the development of applications for the MPPA as it has to contain the system core OS, the 16 PC code and data. In our case, we see in section 4 that it reduces the choices of possible parallelization approaches.

Four I/O interfaces allow communications between the host machine and the clusters for two of them and between clusters and the Ethernet network for the other two, but with the current version of the middleware we only have access to one of the I/O interfaces: the software tools designed for the MPPA and grouped under the name of MPPA ACCESSCORE are currently under development. This I/O interface has a quad core SMP processor with a 4GB DDR3 memory and a PCI Gen3 interface for communications with the host. Different sofware connectors are available to implement synchronous or asynchronous communications using a simple buffer or queues.

[Jouandeau, 2013] states that "the computing capabilities of Intel i7 3820 processor with 8 cores and a MPPA processor with 256 cores are close". The estimate is based on the performance observed for a single core on solving the spin glass problem and multiplied by the number of cores, which implies parallel algorithms with zero communication time. Even if the MPPA-256 card seems to be limited in its performance, it is only the first member of the MPPA MANYCORE family. A *Coolidge* processor with 1024 cores is expected in 2015. In addition, several MPPA-256 cards can be used together in the same host machine to increase the computing capacity. Therefore, the MPPA architecture presents possibilities that deserve investigations.

4 Setup of the parallelization on the MPPA

Our experiments were carried out on a server equipped with an Intel Core I7 at 3.6GHz running Linux OS and a PCIe Application Board AB01 equipped with a chip MPPA-256 [Kalray, 2013].

Our program is based on Jean-Noël Vittaut's *LeJoueur*, written in C++. This player uses Prolog to realize a fast instantiation of the game rules [Vittaut and Méhat, 2014] and creates a propnet. The propnet logic circuit is factorised to reduce its size and each layer of logic gates of the circuit is then translated into a set of rules that can be evaluated very quickly with binary operators. This is the set of rules that is sent to the MPPA to explore the game.

The limited memory inside the clusters implies a coding as concise as possible so that the *propnet* can fit into it. Unfortunately, for the most complex games, like Hex, the size of the *propnet* exceeds the available space.

The size of the remaining memory inside the clusters does not allow the construction of a game tree of significant size.

Therefore *tree parallelization* inside each individual cluster, *root parallelization* and *UCT-Treesplit* cannot be performed on a MPPA. The only remaining choice is *leaf parallelization*.

For all the experiments presented in this article we used synchronous communications. Thus the benchmark results can later be compared with the ones using asynchronous communications. The host machine send sets of game positions to the MPPA I/O interface which distributes them to the different clusters. The first PC (PC0) of each cluster is responsible for communicating with the I/O and for starting threads on other PCs via the system core. A thread can be executed by each one of the 15 remaining PCs. The PC0 sends received positions to the threads and waits for all the results before sending them back at once to the I/O.

The I/O interface waits for the results from all the clusters before sending them back at once to the host. Each transmission of a position set, calculation of the playouts and recovery of the results (the scores) is referred to as a *run*.

5 Scaling for variable size of communications

At first, we investigate the capacity of the communication networks to transmit game positions, which can present a significantly different size depending on the game at hand, between the host computer and the PCs without an hindering variation of the communication time.

In GGP the size of position descriptions varies significantly from one game description to another in a ratio from one to ten: we need about 200 to 2000 bytes to represent a position for commonly used games. To make one playout from a different position on each PC, the size of a communication has to be multiplied by the number of PCs.

To evaluate the scalability of the MPPA we varied the size of positions sent between the host and the I/O node and between the I/O node and the clusters from 200 to 2000 bytes. To prevent the variable time required to calculate the playouts from disturbing measurements, no playout was actually calculated and an empty result was returned immediately. Each execution of the program performs 1000 runs. We measured performance as the time necessary to execute these 1000 runs for 1 to 16 clusters with 1 to 16 threads per cluster.

Figure 1 presents the experimental results. With one thread on one cluster, the running time is almost constant while the size of communications is multiplied by ten. With 16 threads on one cluster, there is additional time corresponding to the thread initialization and the data distribution among threads. The curve corresponding to one thread on each of the 16 clusters shows the additional time required for the I/O node to set the connector to the destination cluster. Scaling is then constrained by two aspects: first, when the I/O node receives a set of positions it needs time to distribute them among the clusters and, second, the thread start time inside a cluster is not negligible.

Scaling is only slightly hindered when considering these two aspects separately but it is not the case when we combine them. The curve for 16 clusters and 16 threads per cluster shows a significant degradation of the scaling. However, we see that scaling remains acceptable since the execution time is multiplied by about 5 when the message size is multiplied by 10.

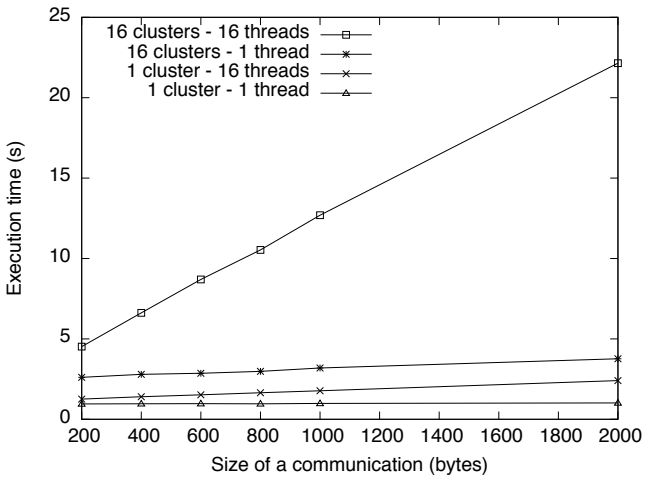

Figure 1: Evolution of the execution time depending on the size of communications (1000 runs).

6 Playout calculation approaches

In the restrictive conditions of a *leaf parallelization*, the choice remains of the playout calculation approach. All the PCs of one cluster can be used to speed up each playout by distributing the evaluation of the propnet layers or each PC can perform its own private playout evaluating all the layers by itself.

We conducted these experiments on three games: *Tictactoe* which has short playouts (between 5 and 9 moves) and a small *propnet* which is quick to evaluate, *EightPuzzle* which has a small *propnet* but playouts up to 60 moves and *Breakthrough* which has a *propnet* taking longer to evaluate. Each execution of the program performs 10000 runs. We measured the performance based on the total number of playouts carried out per second.

6.1 Parallelization of propnet evaluations

We first consider the distribution of the propnet layers evaluation between the PCs; the rules of each layer can be evaluated in any order but each layer depends on its predecessor and a synchronization is needed.

Figure 2 presents the performance obtained on the game of *Tictactoe* for 1 to 16 clusters with 1 to 16 threads per cluster. The different curves confirm that increasing the number of active clusters does improve the number of playouts per second and has no impact on the variation of performance. The latter depends on the number of threads used in the clusters. Each curve shows a large degradation as the evaluation of propnet layers is distributed between the threads. It demonstrates the significant synchronization cost generated by the division of playout computation.

We conclude that the distributed computing of a playout cannot provide any benefit considering the overhead introduced by the synchronization barrier between layers.

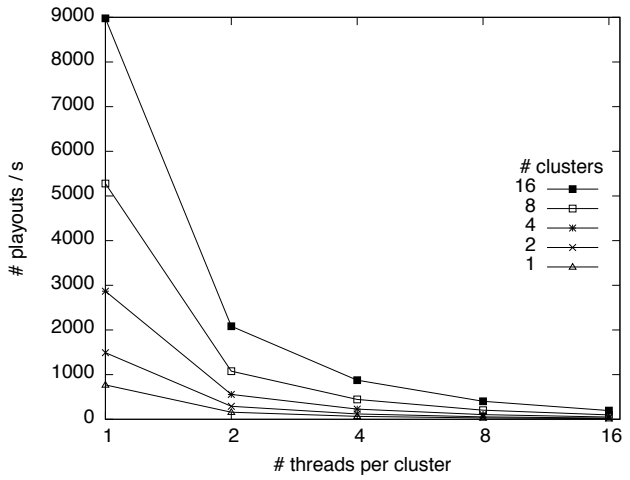

Figure 2: Evolution of the mean number of playouts per second for 10000 runs of the game of *Tictactoe* when a playout calculation is distributed over threads.

6.2 Parallelization of playouts

To avoid the cost generated by the inter-layer synchronization, it is possible to have every PC perform its own private playout, layer after layer. Synchronization is only needed when the initial positions are set and when the results are gathered.

Figure 3 presents the results for the three games with 1 to 16 clusters and 1 to 16 threads per cluster. It shows that the performance scales well.

Results for the game of Tictactoe show a progression from $\approx$450 playouts/s for one cluster with one thread to $\approx$77700 playouts/s for 16 clusters with 16 threads, i.e. a multiplication by 170 of the number of playouts per second for a multiplication by 256 of the computing power.

For the game *EightPuzzle*, the number of playouts is multiplied, at best, by 133 (for 14 threads per cluster) using 224 PCs. The performance does not scale well after 14 threads. We explain this result by the constant length of playouts: playing randomly, the solution has little chance of being found and playouts are stopped after 60 moves by the stepper. Therefore, all the scores are sent to the I/O node at the same time by all clusters causing a congestion of the communication network. A similar slowdown hindering scalability can be observed in Tictactoe by forcing the players to completely fill the grid for each game: playout length is then always set to 9 moves.

Scaling is better for *Breakthrough*. Computation time is longer, therefore communication and synchronization times are lower in comparison. The number of playouts is multiplied by 155 for a multiplication by 256 of the computing power.

7 Thread management

In the previous experiment threads were restarted for each set of playouts. Another strategy is to keep the threads alive waiting for playout requests. To test different synchronization

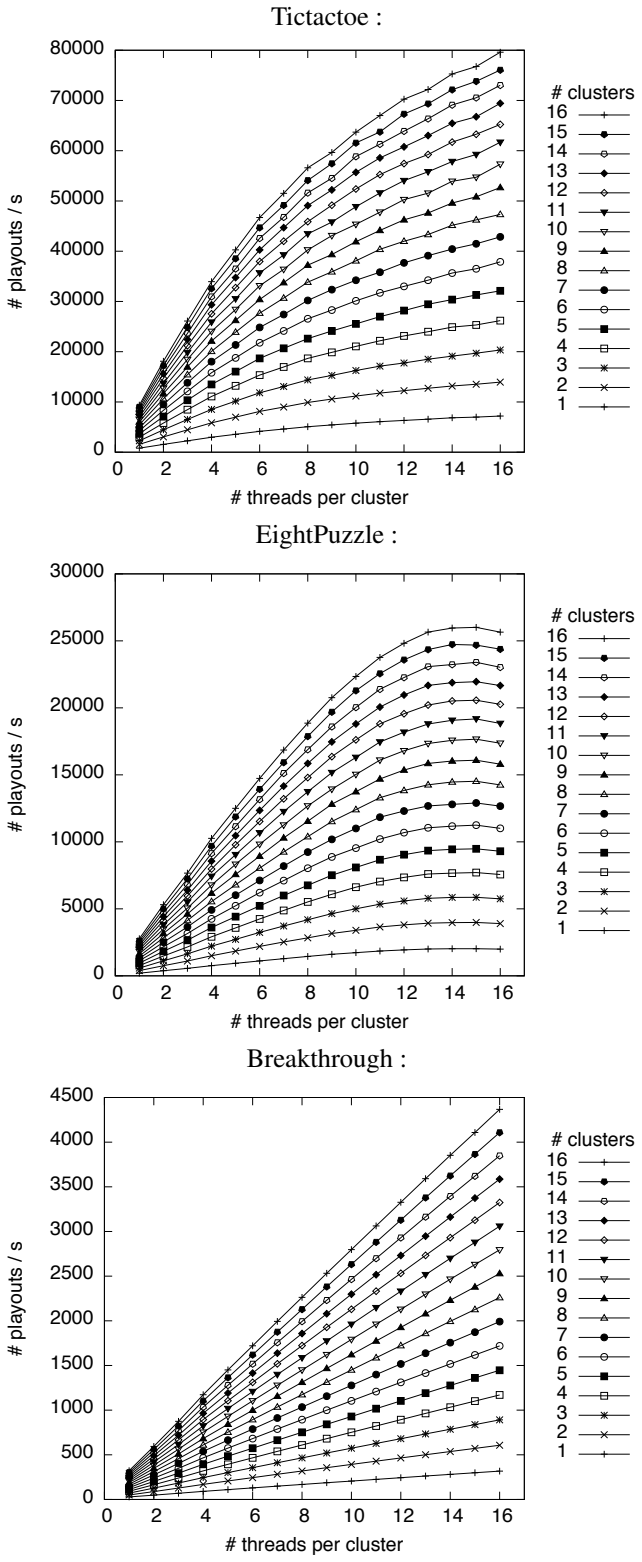

Figure 3: Evolution of the number of playouts per second (average over 10000 runs) with the calculation of one playout per thread.

approaches, we changed the thread management in the setting of the experiment of section 6.2: on each PC of each cluster, a thread is started waiting for a position.

All the positions sent were initial positions of the game and the id of each PC was used as a random seed for the random playouts it executed. The tests were performed on different games, varying the number of clusters and number of threads per cluster; 10000 runs was executed every time.

In the first part of the experiment we used the Posix functions `pthread_cond_wait` and `pthread_cond_signal` to communicate computation requests to the threads. In the second part, we replaced the Posix functions by a busy wait. In the third part we used portal connectors which are part of the MPPA ACCESCORE tools and use the NoC to create communication paths between the PC0 thread and the other PC threads of the cluster. This third approach avoids the use of mutexes.

Results are displayed in figure 4 for 16 clusters and 1 to 16 threads per cluster and compared to the results obtained with restarted threads. For the games *EightPuzzle* and *Breakthrough*, we compare only the restart of the threads with the use of the Posix functions `pthread_cond_wait` and `pthread_cond_signal` because they were sufficient for conclusive results.

With the game of *Tictactoe*, the use of the portal is less effective regardless of the number of threads used, and the different approaches keeping threads alive do not scale well above 14 threads per cluster. We note with surprise that restarting threads for each request of calculation provides better scalability and gives the best results. The results obtained on the games *EightPuzzle* and *Breakthrough* do not show such a marked difference since the curves are nearly identical. It confirms that keeping threads alive provides no benefit.

This can be explained by the use of a memory shared by all the PCs of a cluster and the fact that all the synchronization primitives end on a spinlock or equivalent. When a thread has finished its work, it waits for a signal by polling in the shared memory. Therefore, if the other threads have not finished the calculation of their playout, they are slowed down since they also need to access the shared memory. The main process running on PC0 and responsible for the communications with the I/O node is also slowed down. The more threads finish their task, the slower execution of other threads is. On the contrary, when a thread uses `pthread_exit` the PC running this thread is placed in an idle state and does not disturb the work of other PCs. Halting and relaunching the threads is then more efficient.

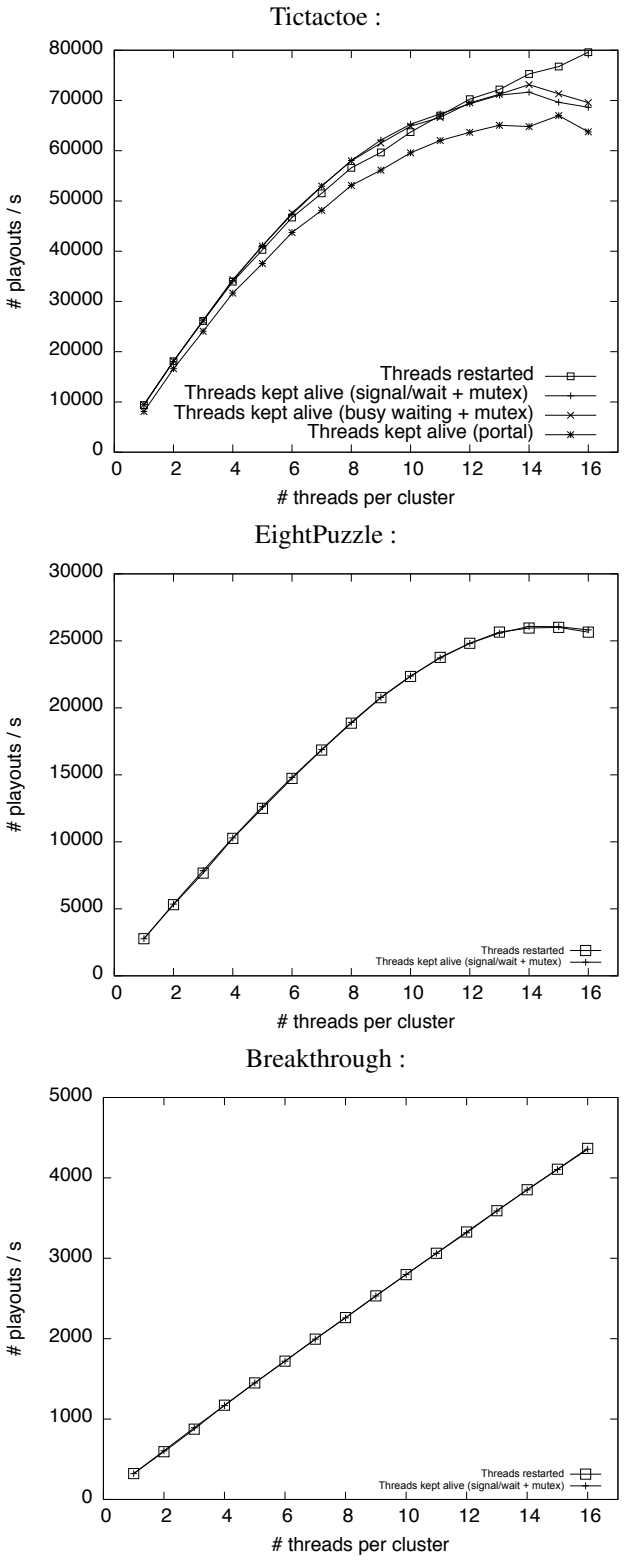

Figure 4: Evolution of the performance i.e. number of playouts per second (10000 runs, 16 clusters) with different thread management modes.

8 Conclusion

In this paper we have studied the capabilities offered by a *Multi Purpose Processor Array* (MPPA) architecture for the parallelization of MCTS algorithms in the field of *General Game Playing*. The limitation of the memory inside the cluster to a size of 2MB is a major constraint. Among the various parallelization techniques described in the literature the only applicable one, with this limited memory space, is *leaf parallelization*.

We have demonstrated that the MPPA provides good scal-

ability when increasing the size of communications, giving good results when using synchronous communications and sending large sets of game initial positions to be processed in a single run.

We have considered two approaches for the calculation of playouts. The distributed computing of a playout on the different PCs of a cluster causes an important synchronization overhead. The calculation of a complete playout per PC gives better results.

We were able to establish that on MPPA it is more efficient to restart threads for each calculation request. All synchronization primitives on the MPPA end on a spinlock. Therefore, idle threads kept alive slow down the working ones because of memory access competition.

It would be possible to reduce the weight of communications by making several playouts from each sent position, but carrying out several playouts from the same position may be less beneficial for the UCT exploration than starting from different positions [Chaslot et al., 2008].

One may think that the use of asynchronous communications would improve these results but unfortunately the first experiments we have conducted have yielded disastrous results. This comes from middleware problems that our experiments have brought to light. These problems should be fixed by Kalray in the next release. The use of asynchronous communications will therefore be the subject of future experiments.

We will also compare our results with what could be achieved with a GPU using a framework like Cuda on the same problem. The SIMT architecture requires the use of synchronous operations but this can be an advantage to distribute the calculation of each layer of the *propnet* without the need for a specific synchronization barrier. Moreover, the large quantity of memory shared by computing units allows the implementation of different parallelization techniques.

Future works also include the test of new approaches for MCTS parallelization. For example, the creation of mini-trees in cluster memory can save communication costs.

The SHOT alternative to UCT [Cazenave, 2015] offers interesting perspectives as it scales well in addition to using less memory and it can be efficiently parallelized.

The work presented here is an updated version of [Hufschmitt *et al.*, 2015].

References

[Browne *et al.*, 2012] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo Tree Search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[Cazenave and Jouandeau, 2007] Tristan Cazenave and Nicolas Jouandeau. On the parallelization of UCT. In *Proceedings of the Computer Games Workshop*, pages 93–101, 06 2007.

[Cazenave and Jouandeau, 2008] Tristan Cazenave and Nicolas Jouandeau. A Parallel Monte-Carlo Tree Search Algorithm. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 72–80. Springer, 2008.

[Cazenave, 2015] Tristan Cazenave. Sequential Halving Applied to Trees. In *IEEE Trans. Comput. Intellig. and AI in Games*, volume 7, pages 102–105, 2015.

[Chaslot *et al.*, 2008] Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel Monte-Carlo Tree Search. In *Computers and Games*, pages 60–71, 2008.

[Enzenberger and Müller, 2010] Markus Enzenberger and Martin Müller. A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm. In *Proceedings of the 12th International Conference on Advances in Computer Games*, ACG'09, pages 14–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[Finnsson, 2012] Hilmar Finnsson. *Simulation-Based General Game Playing*. Doctor of philosophy, School of Computer Science, Reykjavík University, 2012.

[Gelly *et al.*, 2006] Sylvain Gelly, Yizao Wang, Rmi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo go. Technical Report 6062, Inria, 2006.

[Gelly *et al.*, 2008] Sylvain Gelly, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, and Y. Kalemkarian. The Parallelization of Monte-Carlo Planning - Parallelization of MC-Planning. In Joaquim Filipe, Juan Andrade-Cetto, and Jean-Louis Ferrier, editors, *ICINCO-ICSO*, pages 244–249. INSTICC Press, 2008.

[Graf *et al.*, 2011] Tobias Graf, Ulf Lorenz, Marco Platzner, and Lars Schaefers. Parallel Monte-Carlo Tree Search for HPC Systems. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 365–376, Berlin, Heidelberg, 2011. Springer-Verlag.

[Hufschmitt *et al.*, 2015] Aline Hufschmitt, Jean Méhat, and Jean-Noël Vittaut. Using MPPA architecture for UCT parallelization. In *Proceedings of the 8th International Conference on Game and Entertainment Technologies*, 2015.

[Jouandeau, 2013] Nicolas Jouandeau. Intel versus MPPA. Technical report, LIASD Universit Paris8, 11 2013.

[Kalray, 2013] Kalray. *MPPA ACCESSCORE 1.0.1 - POSIX Programming Reference Manual - KETD-325 W08*. Kalray SA, 07 2013. 142 pages.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[Méhat and Cazenave, 2011a] Jean Méhat and Tristan Cazenave. A Parallel General Game Player. *KI*, 25(1):43–47, 2011.

[Méhat and Cazenave, 2011b] Jean Méhat and Tristan Cazenave. Tree parallelization of Ary on a cluster. In *GIGA 2011, IJCAI 2011*, Barcelona, 07 2011.

[Schaefers *et al.*, 2011] Lars Schaefers, Marco Platzner, and Ulf Lorenz. UCT-Treesplit - Parallel MCTS on Distributed Memory. In *MCTS Workshop*, Freiburg, Germany, 06 2011.

[Schiffel and Björnsson, 2014] Stephan Schiffel and Yngvi Björnsson. Efficiency of GDL Reasoners. In *IEEE Trans. Comput. Intellig. and AI in Games*, volume 6, pages 343–354, 2014.

[Schkufza *et al.*, 2008] Eric Schkufza, Nathaniel Love, and Michael R. Genesereth. Propositional Automata and Cell Automata: Representational Frameworks for Discrete Dynamic Systems. In *AI 2008: Advances in Artificial Intelligence, 21st Australasian Joint Conference on Artificial Intelligence, Auckland, New Zealand, December 1-5, 2008. Proceedings*, pages 56–66, 2008.

[Soejima *et al.*, 2010] Yusuke Soejima, Akihiro Kishimoto, and Osamu Watanabe. Evaluating Root Parallelization in Go. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):278–287, 2010.

[Vittaut and Méhat, 2014] Jean-Noël Vittaut and Jean Méhat. Fast Instantiation of GGP Game Descriptions Using Prolog with Tabling. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pages 1121–1122, 2014.